AFRL-VA-WP-TP-2002-319

TRANSITIONING TO PC-BASED SIMULATION–ONE PERSPECTIVE

Joseph Nalepka, Thomas Dube, Glenn Williams, Adam Snyder, Thomas Danube, and G. Jeff Slutz

JUNE 2002

Approved for public release; distribution is unlimited.

AIR VEHICLES DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
June 2002	Preprint	Final, 08/01/2001 – 05/01/2002

4. TITLE AND SUBTITLE

TRANSITIONING TO PC-BASED SIMULATION–ONE PERSPECTIVE

5a. CONTRACT NUMBER
In-house

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Joseph Nalepka, Thomas Dube, Glenn Williams and Adam Snyder
(AFRL/VACD)
Thomas Danube (L3 Communications)
G. Jeff Slutz (Protobox)

5d. PROJECT NUMBER
232F

5e. TASK NUMBER
IH

5f. WORK UNIT NUMBER
PA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Simulation and Assessment Branch (AFRL/VACD)
Control Sciences Division
Air Vehicles Directorate
Air Force Research Laboratory, Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7542

L3 Communications, Inc.

Protobox LLC

8. PERFORMING ORGANIZATION REPORT NUMBER
AFRL-VA-WP-TP-2002-319

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AIR VEHICLES DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/VACD

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TP-2002-319

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Conference paper to be published August 5, 2002 in *Proceedings of AIAA Modeling and Simulation Technologies Conference*. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Doing "more for less" is a recurring theme facing researchers in many simulation facilities. In the area of simulation, one way to reduce operational costs is to invest in and operate inexpensive simulation computer systems. Traditionally, this has not been possible because very specialized computer systems were required to build simulation architectures that provided a deterministic timing mechanism and also assured that simulation processes executed in a predetermined order throughout the entire execution period of the simulation. However, with the advent of 2.0 GHz processors, powerful graphics cards, and the wide availability of software, the personal computer (PC) is now becoming a realistic option for developing real-time simulation architectures. Air Force Research Laboratory (AFRL) researchers at the Aerospace Vehicle Technology Assessment and Simulation (AVTAS) Laboratory have integrated and tested a simple, PC-based, real-time simulation framework that executes under the Linux operating system. Using various hardware- and software-based timing mechanisms, along with nonspecialized software tools, the AVTAS Laboratory has developed a real-time simulation architecture that will enable the execution of simulation experiments on dual-CPU and quad-CPU PCs.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Joseph Nalepka
Unclassified	Unclassified	Unclassified	SAR	14	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 904-6547

AIAA-2002-4863

TRANSITIONING TO PC-BASED SIMULATION – ONE PERSPECTIVE

Joseph Nalepka, Thomas Dube, Lt, USAF, Glenn Williams, Adam Snyder
Air Force Research Laboratory

Thomas Danube
L3 Communications, Inc.

G. Jeff Slutz
Protobox LLC

ABSTRACT

Doing "more for less" is a recurring theme facing researchers in many simulation facilities. In the area of simulation, one way to reduce operational costs is to invest in and operate inexpensive simulation computer systems. Traditionally, this has not been possible because very specialized computer systems were required to build simulation architectures that provided a deterministic timing mechanism and also assured that simulation processes executed in a predetermined order throughout the entire execution period of the simulation. However, with the advent of 2.0 GHz processors, powerful graphics cards and the wide availability of software, the Personal Computer (PC) is now becoming a realistic option for developing real-time simulation architectures. Air Force Research Laboratory (AFRL) researchers at the Aerospace Vehicle Technology Assessment and Simulation (AVTAS) Laboratory have integrated and tested a simple, PC-based, real-time simulation framework that executes under the Linux operating system. Using various hardware and software-based timing mechanisms, along with non-specialized software tools, the AVTAS Laboratory has developed a real-time simulation architecture that will enable the execution of simulation experiments on dual-CPU and quad-CPU PCs.

INTRODUCTION

One of the challenges in today's laboratory environment is trying to do more work with fewer resources. These resources include people, money, and simulation hardware to name a few. Traditionally, simulation engineers were required to purchase and integrate very specialized and vendor specific computer systems into their simulation facilities. The primary reason for this is that these computer systems were developed specifically for real-time applications and were able to provide the accurate timing mechanisms that were necessary for synchronizing time critical simulation processes.

Although specialized computing systems are excellent for providing a deterministic timing source and real-time architecture, they do come with several drawbacks. The first and most obvious drawback is cost. These specialized computer systems tend to have very unique architectures and operating characteristics that cause their costs to be significantly more than other "conventional" computer systems. Second, because these are specialized computer systems, many of the hardware and software components used with these machines are only developed and maintained by the vendor. Consequently, the user, in most cases, is required to purchase a maintenance agreement with the vendor that will cover service, repairs, and updates to both the computer hardware and software. Third, in some cases, the operating system for these computer systems is unique and thus a significant "learning curve" must be overcome in order to understand and utilize the system properly. Fourth, many software development and troubleshooting tools are not compatible with these specialized computer systems. Consequently, a facility must resort to purchasing these tools from the vendor at a large cost or develop these tools on their own. Finally, because of the specialized architecture, many of the simulation software models have to be modified, compiled, or

run in a manner that takes advantage of the real-time operating features of the computer system. The unfortunate result of these modifications is that many simple software modules become complicated and less portable to other real-time computer systems.

Although specialized real-time computer systems have been the status quo for simulation, they are now receiving stiff competition from another computation platform: the Personal Computer (PC). There are several reasons for this new competition. First, with processor speeds now approaching the 2 GHz plateau, PCs now have the ability to execute the most complicated and computationally demanding software models very quickly. Second, PCs are relatively inexpensive when compared to specialized real-time computer systems. Third, an enormous amount of software is widely available for PCs, some of which can be obtained for little or no cost from the Internet or other "shareware" sources. Fourth, with the current advances in graphics card technology, PCs are becoming more and more capable of rendering very complex graphical images and scenes. Finally, because there are many PCs in use today, it is very easy to find a vendor that can troubleshoot and repair various hardware and software problems without the need for purchasing a hardware or software maintenance agreement from the computer manufacturer itself.

One of the major drawbacks of using PCs for real-time simulation is their inability to provide a deterministic timing source or to guarantee execution order for the simulation processes. Researchers in the Aerospace Vehicle Technology Assessment and Simulation (AVTAS) Laboratory at the Air Force Research Laboratory (AFRL) are addressing this drawback and have successfully implemented a PC-based, real-time simulation architecture using the Linux operating system. The purpose of this paper is to quickly review previous results and then present the current research findings concerning the use of Linux for real-time simulation architectures.

REAL-TIME SYNCHRONIZATION

One of the most critical features of PC-based, real-time, simulation architectures is the ability to utilize an accurate or "deterministic" timing source. A deterministic timing source is one in which a particular simulation frequency or frame rate is maintained throughout the entire execution phase of the simulation, regardless of the activity that may be occurring on the computer system. This type of real-time performance is necessary when there is a desire to minimize latency within a simulation or when simulation specific models are required to update at a specific and constant rate.

Before proceeding with the latest results and innovations for deterministic timing under Linux, it is first necessary to summarize previous results. For the AVTAS Laboratory, and as described in detail in Reference 1, it was necessary to have a simulation architecture that could provide deterministic timing to within one millisecond of the desired simulation frequency. The first approach taken for achieving this requirement was to use a set of Institute of Electrical and Electronics Engineers (IEEE) software standards called POSIX. Within POSIX, there exists a timing mechanism called the interval timer that provides the simulation engineer with a software-based, programmable timing source. This timer is a counter that is decremented after each computer clock tick. The initial value of this timer is based upon the user specified update frequency. When this counter reaches zero, the timer is said to "expire" and a signal is sent to a process that is waiting to capture this signal. The timer is then automatically reset and the countdown to the next interval begins.[2]

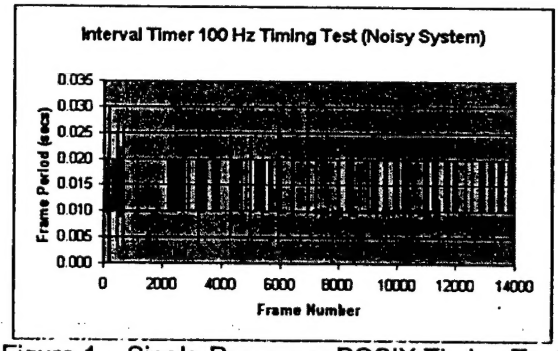

Figure 1 – Single-Processor POSIX Timing Test (Noisy System)

Unfortunately, it was discovered that the POSIX interval timer was easily influenced by background system activity[1]. Figure 1 and Figure 2 show timing results for the POSIX based interval timer on both a "noisy" single-CPU and "noisy" quad-CPU computer system. A noisy computer system is one in which various system activities such as disk searching, software compilation, mouse movement, and various window operations are occurring in order to evaluate the timer under very demanding system conditions. What these timing results show is that there is considerable variation in the timing mechanism for a 100 Hz simulation frequency, even on a system that has 4 CPUs

available for data processing. From these results, it was concluded that the POSIX interval timer (under Linux) would not be an acceptable timing source for the AVTAS Laboratory real-time simulation architecture.

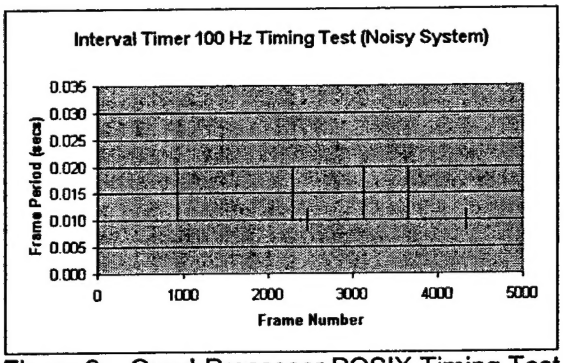

Figure 2 – Quad-Processor POSIX Timing Test (Noisy System)

One thing that should be noted is that the POSIX interval timer is an example of a "non-polling" timer. A non-polling timer allows the calling process to "sleep" and thus yield the processor to another process, while the calling process waits for an alarm signal from the operating system. The greatest advantage to this type of timer is that the processor can be used for other processing and the calling process can be "woken up" at the appropriate time. However, research at the AVTAS Laboratory has yielded some surprising results from "polling" timers, which constantly sample a variable or memory address until it reaches a specified value.

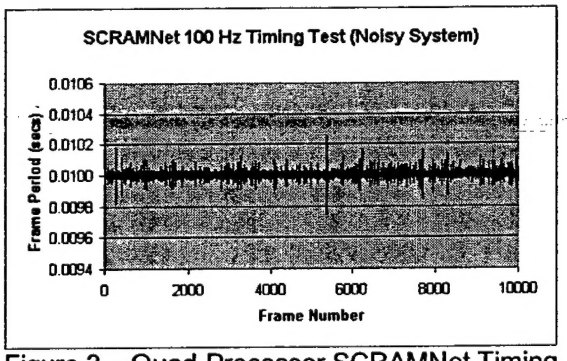

Figure 3 – Quad-Processor SCRAMNet Timing Test (Noisy System)

The second approach that was taken by AFRL researchers for simulation timing was a hardware-based, polling timer solution: SCRAMNet. Typically, for simulation experiments, SCRAMNet is used as a shared memory extension to allow different simulation computers to share simulation data. However, an additional feature of SCRAMNet is its user-programmable timer that has a resolution of 1.706 microseconds. Figure 3 shows a timing diagram that validates the utility of SCRAMNet as a simulation timing mechanism. Even under noisy computer system conditions, fluctuations of the SCRAMNet based timer were well within the one-millisecond resolution specification for the AVTAS Laboratory simulation architecture. From these results, it was concluded that the SCRAMNet timer would be an acceptable timing source for the AVTAS Laboratory real-time simulation architecture.

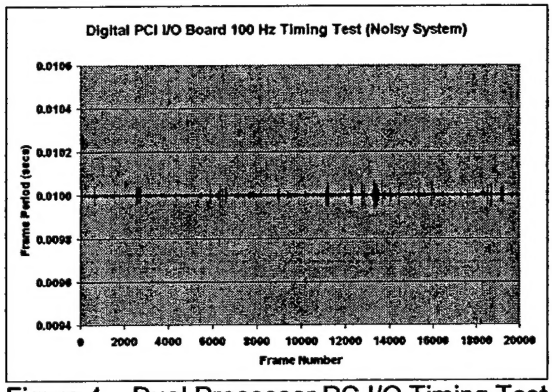

Figure 4 – Dual Processor PC I/O Timing Test (Noisy System)

Although SCRAMNet was demonstrated to be a very reliable timing mechanism, its unfortunate drawback is that it can cost more than the PC itself (nearly $6000). Consequently, this makes SCRAMNet a very expensive acquisition, especially if one is only considering using it as a simulation timing mechanism. As a result, AFRL researchers began exploring two alternative solutions for simulation timing. The first solution was a hardware-based implementation that was significantly less expensive than a SCRAMNet board (less than $100). This implementation was a 24-Channel, PCI Digital I/O Board with programmable timer (2 microsecond resolution) that can be used as both a polling and non-polling timer. As was done in previous experiments, timing measurements were also made using this I/O Board. These measurements were done on a dual-CPU, 1.0 GHz PC with 1 GB of RAM at a timing frequency of 100 Hz under noisy computer system conditions. Although not presented here, when this board was used as a non-polling timer, the performance was not acceptable. When used as a polling timer, however, it functioned very much like the SCRAMNet timer. These results are shown in Figure 4. From these results, it is seen that this timer is well within the one-millisecond resolution specification for the AVTAS Laboratory

simulation architecture. Consequently, it was concluded that the Digital I/O Board would be an acceptable timing source for the AVTAS Laboratory real-time simulation architecture provided it was used in a polling mode only.

The second alternative solution that was explored was a "purer" software-based, polling timer solution. This software solution uses a standard Linux utility called *gettimeofday*. The purpose of this utility is to retrieve the time since Epoch, or, in more common terms, the time in seconds and microseconds since 1 January 1970[3]. This timer is implemented by first using *gettimeofday* at the start of a simulation frame to get a reference time for the frame. The simulation software continually reads or "polls" the system clock using *gettimeofday* until it is time for the next simulation frame to begin. Before the next frame is started, the frame reference time is incremented by the absolute simulation frame period and the countdown until the start of the next simulation frame is begun using this new reference frame time.

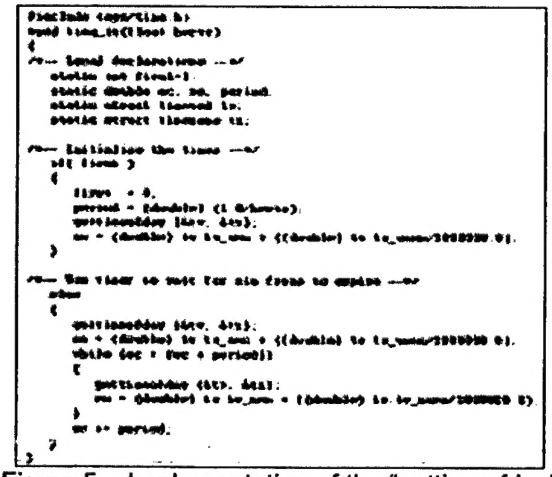

Figure 5 – Implementation of the "gettimeofday" Timing Mechanism

The main reason for incrementing time based on the absolute simulation frame period is to minimize the long-term timing errors of the simulation. Because of round off and other numerical precision errors, the frame period, when measured "absolutely" using *gettimeofday*, may vary slightly between two successive frames. However, these small errors between simulation frame times will become additive and result in a very large timing error at the end of the simulation. By adding the absolute frame period to the previous reference time, it is guaranteed that the long-term simulation timing error is reduced to the error as measured between the last two simulation frames, which is very small. An example of the *gettimeofday* polling timer implementation is shown in Figure 5.

Timing measurements were conducted on this timing implementation using a dual-CPU, 1.0 GHz PC with 1 GB of RAM at a timing frequency of 100 Hz under noisy computer system conditions. The results of these measurements are shown in Figure 6. Despite being a software-based timer, these results show surprisingly good accuracy in maintaining the desired simulation frequency. As expected, additional computer system activity had an influence on the accuracy of this timer, but surprisingly it performed only slightly worse than either of the "hardware-based" implementations. This reduced performance still produced a simulation frequency that was well within the one-millisecond resolution specification for the AVTAS Laboratory. As a result, it was concluded that this timing mechanism would also be an acceptable timing source for the AVTAS Laboratory real-time simulation architecture.

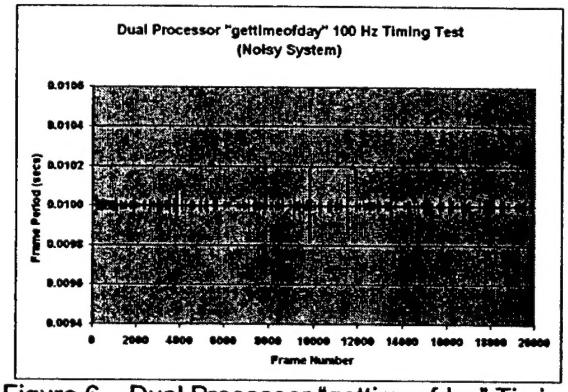

Figure 6 – Dual Processor "gettimeofday" Timing Test (Noisy System)

What has been demonstrated thus far is that there are several alternatives available for simulation timing if small amounts of variation are tolerable with the timing mechanism itself. For a low cost solution, both the software-based timer using the *gettimeofday* function and the Digital I/O Board provide an excellent solution. However, if it is necessary to share data amongst different computer systems, the SCRAMNet solution, although more expensive, is able to provide the simulation with not only a shared memory mechanism between the computer systems but also a very reliable timing source. The lesson to be learned from these timing experiments is that there are a number of simulation timing solutions available for a simulation when utilizing the Linux operating system. However, the choice of timing mechanism for a particular real-time architecture will be based on

the timer's resolution requirements, the amount of acceptable variation in the timing source from the desired frame rate, and the cost constraints of the simulation facility.

REAL-TIME ARCHITECTURE DESIGN

A key component of any real-time simulation architecture is the ability to control those processes that are running within this architecture. For instance, a requirement of the AVTAS Laboratory real-time architecture is the ability to execute simulation processes in a specific order at a specific simulation rate. In the case when multiple CPUs are required, it is necessary for these simulation processes to execute in parallel to one another to assure maximum computational efficiency. A graphical representation of this design philosophy is shown in Figure 7. In this figure, each bar represents a real-time process and the length of each bar represents the amount of time required by the real-time process to execute within a simulation frame.

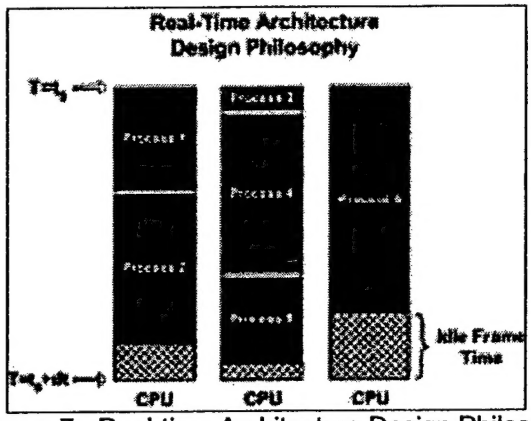

Figure 7 - Real-time Architecture Design Philosophy

One of the problems with this approach, however, is that it is not completely supported under POSIX. With POSIX, the user has the ability to assign an execution priority to a process in order to control execution order, however, it does not provide a mechanism for assigning processes to a specific processor. Consequently, the operating system decides which CPU will run a real-time simulation process based on CPU availability at a given instant of time. Because the POSIX standard did not provide a mechanism for achieving this, an alternative solution, developed by Tim Hockin, (http://isunix.it.ilstu.edu/~thockin/pset) was implemented. The purpose of these utilities, called PSET, is to enable the user to assign a process to a specific processor, restrict a processor's ability to execute processes, restrict a processor from running at all, and get information about a processor's execution state. Use of these utilities simply involves patching them into the Linux kernel and then rebuilding the kernel itself.

At the time of its implementation into the AVTAS Laboratory in 2001, this simple operating system patch along with the POSIX standards enabled AFRL researchers to create a real-time, concurrent processing simulation architecture. Unfortunately, this implementation had two major drawbacks. First, because this implementation requires the use of specialized function calls within the real-time simulation architecture (as a result of the PSET utilities), this architecture itself became less portable between PCs. Although this problem is easily resolved by installing the patch onto all PCs within the AVTAS Laboratory, it is still a level of effort that should not be required. Second, and probably most bothersome, is that the PSET utilities themselves are not very portable between different versions of the Linux Kernel. The initial development of the PC-based architecture within the AVTAS Laboratory was done under Linux Kernel Version 2.2. As new computers and new computer hardware were acquired for the AVTAS Laboratory, it became necessary to upgrade to Linux Kernel Version 2.4. An unfortunate artifact of this upgrade was that the PSET utilities, written for Linux Kernel Version 2.2, no longer operated properly. In order to fix this problem, it was necessary for AFRL researchers to become familiar with the Linux Kernel software itself in order to modify the PSET utilities for proper operation under this newer version of the Linux Kernel. Although this is not an impossible task, the idea of having to perform this same exercise for future Linux Kernel releases made this a very undesirable and high maintenance task for AFRL researchers. Consequently, an alternative solution was sought for creating the real-time, concurrent processing simulation architecture.

For AFRL researchers, the ideal approach to developing the real-time architecture was to eliminate any reliance on "specialized" software for the simulation architecture. For this reason the POSIX based solution was utilized. This approach provided a mechanism for controlling the execution order of simulation processes (by enabling the assignment of an execution priority to a simulation process), however, it did not guarantee which processor would be utilized during execution. The question asked by AFRL researchers at this point was: Is locking a process to a specific processor critical for a real-time architecture?

There are many arguments that can be presented to justify why a real-time simulation process should or should not be locked to a specific computer processor. One such answer in favor of this approach is that you minimize the amount of lost processing time in a given simulation frame as a result of a process switching between processors during execution. However, the counter argument is that the amount of time lost as a result of this "switching" is on the order of microseconds and that unless you are using nearly all of your simulation frame for executing simulation processes, this switching time will be insignificant and not result in overruns in the simulation frame time.

Still another argument in favor of locking a process is that you guarantee that the only thing that will be executing on a specific processor will be the real-time process itself. This is important because it is undesirable to have non-real-time processes, such as those related to compiling or other operating system activity, interfering with or taking away from the overall simulation frame time. The counter argument to this, however, is that if the execution priority of a real-time process is higher than the execution priority of the non-real-time processes, the real-time processes will have the priority for execution and will not be preempted by the non-real-time processes. This last point, however, assumes that you have at least one processor available for operating system activities. Otherwise, the inability to execute operating system processes will have an adverse affect on the overall operation of the computer system itself. It is for this reason that the AVTAS Laboratory simulation architecture requires at least a dual-CPU computer system (one processor for operating system activity and one for the real-time processes)[1].

Upon review of the AVTAS Laboratory real-time architecture requirements, AFRL researchers came to the conclusion that the ability to control the order of execution of processes far outweighed the requirement for executing simulation processes on a specific processor. The rationale for this decision is that as long as all simulation processes finish within the allotted simulation frame time and that they execute in the desired order, it does not matter if a specific simulation process executes on a different CPU from one simulation frame interval to the next. As was previously mentioned, POSIX already provides a mechanism for assigning real-time processes a higher execution priority over non-real-time processes. Consequently, the only remaining challenge was to implement a mechanism for controlling the order of execution of the real-time processes. The implementation that was adopted was to utilize a standard operating system feature called *semaphores*.

In basic terms, a semaphore is a location in memory, which is visible to all processes and whose value can be tested and set with a single instruction. The test and set operation is, as far as each process is concerned, uninterruptible and once started, nothing can stop it. Depending on the result of the test and set operation, one process may have to sleep until the semaphore's value is changed by another process[4]. The values that are tested can be positive or negative, but, typically, are usually either 0 or 1.

The best way to explain the functionality of semaphores is through an example:

> *Let's say you had two processes, both of whose job was to read and write data to a data file, but only one process can have the file open at a time. It is easiest to think of the resource as actually being the semaphore. A value of 1 means the resource is available, and a value of 0 means the resource is unavailable. At the start of execution, the value of the semaphore (the resource) is initialized to 1 to indicate that it is available. The first process tests the "resource" to see if it's available. Since the value is 1, the first process immediately decrements the semaphore by 1. Now the value of the semaphore is 0, which in effect "locks" the data file to every process except the first process. When the second process tries to test the semaphore, it will see the value is 0, which indicates the resource is unavailable, therefore the operating system will place the second process in the wait queue to wait until the resource becomes available again. It is important to note that second process is no longer a running process, but it is a "sleeping" process. Meanwhile, the first process finishes with the data file. The first process will then increment the semaphore value (making the value 1) to indicate that the resource is available again. The operating system will wake the second process up, because the resource is available again (i.e., the semaphore's value becomes greater than or equal to 1). This will continue, thus enabling each process to access the data file one process at a time*[4].

Before explaining how semaphores were incorporated into the real-time simulation architecture, it is first important to define a new concept developed by AFRL researchers. As a means of identifying

sets of simulation processes that must follow one another in their order of execution, the term "Family" was established. Again, this can be best explained by an example:

Referring to Figure 7, Family 1 would consist of Process 1 and Process 2 because Process 2 can only run after Process 1 has completed. Likewise, Family 2 would consist of Process 3, Process 4, and Process 5. Finally, Family 3 would consist of only Process 6. It is the job of the new semaphore scheme to assure that the order of execution within a Family is maintained. In other words, the job of the semaphores, in the case of Family 2, is to assure that Process 4 does not run until Process 3 is finished and, likewise, to assure Process 5 does not run until Process 4 is finished. Because the order of process execution is only defined within a particular Family, the order of execution in one Family has no affect on any of the other Families. For example, again referring to Figure 7, the order of execution and the time when processes start in Family 1 and Family 3 has no influence on the processes in Family 2.

In order to implement the Family concept using semaphores, three different types of semaphores had to be created: Frame Start, Frame End, and Family. The purpose of each type is as follows:

Frame Start: An array of semaphore elements with each element corresponding to a Family. This semaphore array is used to simultaneously start the first family member in each family. This is analogous to a green traffic light; the time when the light turns green signifies the beginning of the frame.

Family: An array of "n" semaphore elements with each element corresponding to a specific process (family member) within a Family. Each element can take on a value of 0 or 1, with 0 indicating that the family member is not free to execute and 1 indicating that a family member is free to execute. Only one element within this n-element array will have a value of 1, indicating that only one family member within a Family is free to execute at any given time. Elements change state from 0 to 1 starting from the first element of the array and proceeding to the n^{th} element of the array. This is analogous to standard, single lane traffic flow at a green light.

Frame End: An array of semaphore elements with each element corresponding to a Family. Each array element can take on a value of 0 or 1, with 1 indicating that all family members within the Family are done executing and 0 indicating that all family members within the Family have not yet finished executing. This operation is analogous to a yellow traffic light, when the first Family finishes execution, and also to a red traffic light, when all Families have finished execution. When the "light" turns red, the timer checks to see how much time is remaining until the end of the frame and takes any appropriate actions such as reporting simulation frame overruns.

It should be noted that the concept of a Family now replaces the notion of a CPU, as was shown in Figure 7. In this Figure, all processes were expected to run in the same order on the same CPU. With the concept of a Family, the order of execution is maintained but the processor in which execution takes place may be different. This will be explained in more detail later.

The best way to illustrate the use of this semaphore scheme and the concept of a Family is through a graphical example, shown in Figure 8. This figure shows a "snapshot" of execution time within a given simulation frame (denoted by the "Current Time" arrow to the left of the diagram). An initial observation is that all of the Families have begun execution. We know this is true because all elements of the Frame Start semaphore are 0. When the frame was started, they were all set to 1, but they were immediately decremented to 0 by the first family member of each Family. We can also observe that Family 1 has finished executing because its corresponding element in the Frame End semaphore is a 1. Neither Family 2 nor Family 3 has completed its execution yet because their respective elements of the Frame End semaphore array are both 0. Also notice in Family 2 that Process 5 is now permitted to run, because its respective element in the Family 2 semaphore array is a 1 and there can be no more than one family member per Family permitted to run at one time. The same is true with Process 8 in Family 3.

As previously mentioned, the idea of a Family now replaces the conceptual idea of running and locking processes to specific processors. Because the PSET utilities are not being utilized, and POSIX does not support processor locking, this new simulation architecture will only guarantee order of execution and not the processor upon which a simulation process will be executing. The processor that is used and which processes are affected

is strictly up to the scheduling mechanism within Linux. In addition, the processor upon which frame to the next. It is for this reason that one more CPU than simulation Family is required on a simulation computer.

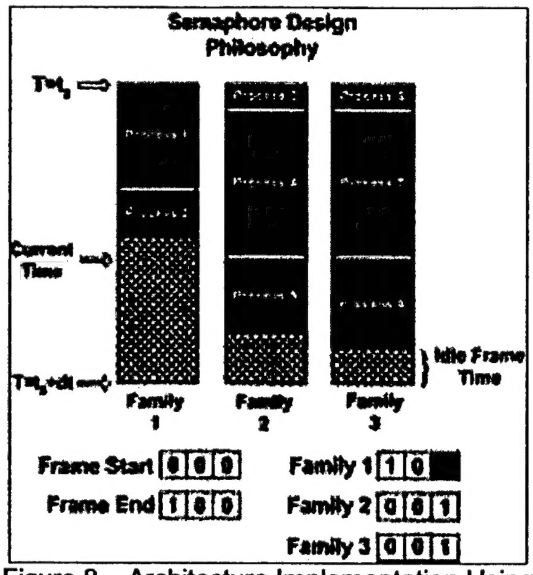

Figure 8 – Architecture Implementation Using Semaphores

FUTURE ENHANCEMENTS

Currently, the AVTAS Laboratory's real-time architecture has been integrated and tested on both dual-CPU and quad-CPU PCs. For many applications, limiting the simulation execution to one PC is not a problem. However, for more complex simulations with a high computational demand, one PC may not be adequate and thus additional PCs may be necessary to augment the demand for more "number crunching" capability. In addition, many facilities, including the AVTAS Laboratory. have computer systems that are different from the PC that can be used for real-time simulations. Consequently, the next step in the evolution of this simulation architecture is to extend the current implementation to work across multiple computer systems and thus create a parallel computing architecture. When this advanced design is implemented, it will provide AFRL researchers with a more flexible and resource efficient mechanism for designing and conducting real-time simulation experiments.

CONCLUSIONS

Researchers at the Air Force Research Laboratory have implemented a simulation architecture that will execute on a Personal Computer using the Linux operating system. This architecture not only provides an accurate, real-time, deterministic update frequency, but is also independent on any specialized software or operating system "patches." Through the use of the POSIX standard, operating system semaphores, and various hardware or software based timing mechanisms, AFRL researchers were able to create a deterministic, real-time simulation architecture that guarantees the order in which simulation processes execute through a newly developed concept called a "Family." Although this architecture makes no guarantees as to which processor a particular simulation process will execute, testing within the AVTAS Laboratory has demonstrated that processor "swapping" does not have a significant impact on the overall performance of the simulation. Consequently, as this real-time architecture is extended to run with more than one computer system, the end result for AFRL researchers will be a low cost and efficient alternative to developing and executing real-time, deterministic simulations.

REFERENCES

1. Nalepka, Joseph, et al., "Real-Time Simulation Using Linux," AIAA 2001-4185, 2001 AIAA Modeling and Simulation Technologies Conference, Montreal, Quebec.

2. Gallmeister, Bill O., "POSIX.4: Programming for the Real World," O'Reilly and Associates, Inc., 1995.

3. Pate, Steve D., "UNIX Internals: A Practical Approach," Addison-Wesley, 1996.

4. Rusling, David A., "The Linux Kernel," David Rusling, 1999.